МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

Курсовая работа

по дисциплине «Алгоритмы и структуры данных»

Тема: исследование хеш-таблицы с цепочками

Вариант 24

Студент гр. 8304 _____ Чешуин Д.И.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург

2019

ЗАДАНИЕ
НА КУРСОВУЮ РАБОТУ

Студент Чешуин Д.И.

Группа 8304

Тема работы: исследование хеш-таблицы с цепочками

Исходные данные:

Написать программу для создания структур данных, обработки и генерации входных данных, использовать их для измерения количественных характеристик хеш-таблицы с цепочками, сравнить экспериментальные результаты с теоретическими.

Содержание пояснительной записки:

- Содержание
- Введение
- Описание структур данных
- Описание алгоритма
- Тестирование
- Исследование
- Исходный код
- Выводы
- Использованные источники

Дата выдачи задания: 11.10.2019

Дата сдачи реферата:

Дата защиты реферата: 25.12.2019

Студент _____ Чешуин Д.И.

Преподаватель _____ Фирсов М.А.

СОДЕРЖАНИЕ

АННОТАЦИЯ

В данной работе была создана программа на языке программирования C++ для генерации, обработки данных и вывода результатов: экспериментальных характеристик хеш-таблицы с цепочками. Результаты были проанализированы и приведены в данном отчёте.

SUMMARY

In this work, a program was created in the C ++ programming language for generating, processing data, and outputting results: experimental characteristics of a hash table with chains. The results were analyzed and presented in this report.

ВВЕДЕНИЕ

В данной курсовой работе реализована структура данных “словарь” на основе хеш-таблицы с цепочечным методом разрешения коллизий, ключом является строка и для получения его хеш-значения используется метод хеширования путём взятия остатка.

ЦЕЛЬ РАБОТЫ

Реализация и экспериментальное машинное исследование алгоритма вставки в хеш-таблицу с цепочками.

1. ОПИСАНИЕ СТРУКТУР ДАННЫХ

Хеш-табли́ца - это структура данных, реализующая интерфейс ассоциативного массива, а именно, она позволяет хранить пары (ключ, значение) и выполнять три операции: операцию добавления новой пары, операцию поиска и операцию удаления пары по ключу.

Хеш-таблица содержит некоторый массив, элементы которого списки пар (хеш-таблица с цепочками). Пример хеш-таблицы с цепочками приведён на рис. 1.

Выполнение операции в хеш-таблице начинается с вычисления хеш-функции от ключа. Получающееся хеш-значение играет роль индекса в массиве. Затем выполняемая операция (добавление, удаление или поиск) перенаправляется объекту, который хранится в соответствующей ячейке.

Ситуация, когда для различных ключей получается одно и то же хеш-значение, называется коллизией. Механизм разрешения коллизий — важная составляющая любой хеш-таблицы.

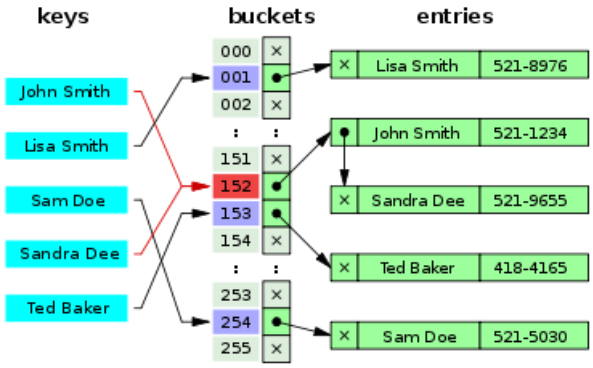

Рисунок 1. Хеш-таблица с цепочками.

2. ОПИСАНИЕ АЛГОРИТМОВ

2.1. Операции поиска, удаления, добавления элемента:

Вставка элемента в хеш-таблицу происходит путём вычисления хеш-значения ключа и последующего добавления элемента в конец списка, находящегося в ячейке массива с номером равным хеш-значению.

Поиск элемента в хеш-таблице осуществляется путём вычисления хеш-значения ключа и последующего сравнения значения ключа со значениями ключей, находящихся в ячейке массива с номером равным хеш-значению. Элемент считается найденным, если будет получено совпадение исходного ключа и ключа пары, хранящейся в списке.

При удалении элемента сначала осуществляется поиск элемента в хеш-таблице как описано выше, и, в случае, если элемент найден, происходит его удаление из списка, находящегося в ячейке массива с номером равным хеш-значению ключа.

2.2. Генерация входных и получение выходных данных

Данные генерируются с увеличением числа элементов на степень двойки начиная с 512 восемь раз. С помощью стандартной 32 разрядной версии вихря Мерсенна – средние случаи и по формуле (k + 1) * 2 ^ i, при этом 2 ^ i должно быть больше генерируемого количества элементов, - худшие случаи. Худшим является случай, когда все элементы добавляются в список, находящийся в одной и той же ячейке массива Для анализа используется среднее время вставки элемента. Среднее время выводится в мс.

3. ОПИСАНИЕ КЛАССОВ

4.1 MyHashTable

```
unsigned hash(const long long int & key) const;
```

Вычисляет хеш ключа методом взятия остатка от деления.

```
void insert(long long int key, ValueT value);
```

Вставляет пару элементов в хеш-таблицу.

```
ValueT get(const long long int& key) const;
```

Получает значение из хеш-таблицу по ключу.

```
void remove(const long long int & key);
```

Удаляет пару элементов по ключу.

```
bool search(const long long int & key) const;
```

Проверяет наличие ключа в таблице.

```
std::string toStr() const;
```

Возвращает строковое представление таблицы.

4. ТЕСТИРОВАНИЕ

Результат запуска программы:

```
Elements - 512 Average case insertion - 0.00390625 ms
Elements - 512 Bad case insertion - 0.00585938 ms
Elements - 1024 Average case insertion - 0.000976562 ms
Elements - 1024 Bad case insertion - 0.00976563 ms
Elements - 2048 Average case insertion - 0.000488281 ms
Elements - 2048 Bad case insertion - 0.0175781 ms
Elements - 4096 Average case insertion - 0.000732422 ms
Elements - 4096 Bad case insertion - 0.0341797 ms
Elements - 8192 Average case insertion - 0.000488281 ms
Elements - 8192 Bad case insertion - 0.0721431 ms
Elements - 16384 Average case insertion - 0.000793457 ms
Elements - 16384 Bad case insertion - 0.140931 ms
Elements - 32768 Average case insertion - 0.00076294 ms
Elements - 32768 Bad case insertion - 0.287572 ms
Elements - 65536 Average case insertion - 0.000854492 ms
Elements - 65536 Bad case insertion - 0.718034 ms
```

5. ИССЛЕДОВАНИЕ

Составим таблицу результатов эксперимента и построим графики зависимости среднего времени вставки от их количества(табл. 1):

Таблица 1. Результаты эксперимента.

Количество вставок	Средний случай	Худший случай	Отношение текущего и предыдущего времени вставки
512	0.00390625	0.00585938	-
1024	0.000976562	0.00976563	1.67
2048	0.000488281	0.0175781	1.80
4096	0.000732422	0.0341797	1.94
8192	0.000488281	0.0721431	2.11
16384	0.000793457	0.140931	1.95
32768	0.00076294	0.287572	2.04
65536	0.000854492	0.718034	2.50

В теории в среднем случае время затраченное на вставку не зависит напрямую от количества элементов. В проведённом эксперименте на случайном наборе данных так же отсутствует какая-либо корелляция между количеством вставок и средним временем вставки, что можно увидеть в табл. 1.

В худшем же случае, согласно теории, время затраченное на вставку линейно растёт при увеличении количества вставляемых элементов. Обращаясь к табл. 1, при увеличении количества вставок в 2 раза, среднее время требуемое на 1 вставку так же увеличивается в среднем в 2.0014 раза, что соответствует теории. Графики зависимостей смотри на рис. 2 и рис. 3.

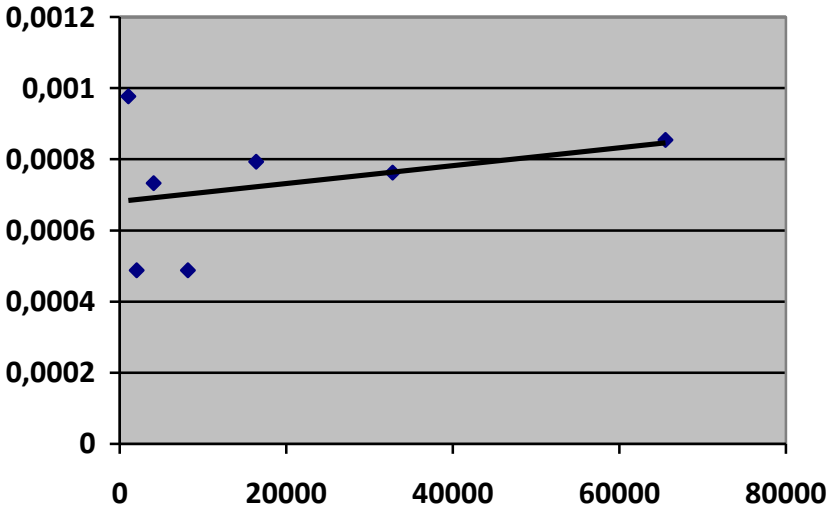

Рисунок 2. Теоретическая и практическая зависимость времени вставки от количества элементов в среднем случае.

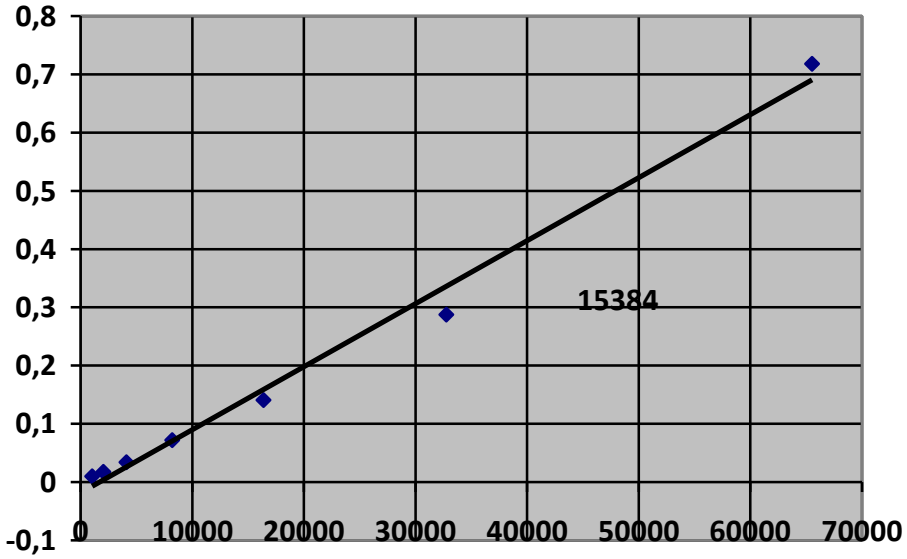

Рисунок 3. Теоретическая и практическая зависимость времени вставки от количества элементов в худшем случае.

Заключение

В ходе работы была написана программа для генерации данных, создания такой структуры данных как хеш-таблица с цепочками и методы для

взаимодействия с ней. С помощью программы были найдены экспериментальные числовые характеристики хеш-таблицы при вставке, при исследовании которых были подтверждены теоретические данные.

Список используемой литературы

1. ru.wikipedia.org/wiki/ Хеш-таблица

Приложение А. Исходный код программы.

main.cpp

```
#include <iostream>
#include <string>
#include <random>
#include <ctime>
#include "myhashtable.h"
```

```
int main()
{
    std::mt19937 randomGen(static_cast<unsigned>(time(nullptr)));
    HashTable<int> table;
    int testsCount = 512;

    for(int k = 0; k < 8; k++){
        clock_t clocks = 0;
        float time = 0;
        float averageTime = 0;

        table.clear();

        for(int i = 0; i < testsCount; i++)
        {
            long long int key = randomGen();
            int value = static_cast<int>(randomGen());

            clocks = clock();

            table.insert(key, value);

            clocks = clock() - clocks;
            time += static_cast<float>(clocks) / CLOCKS_PER_SEC;
        }
        averageTime = time * 1000 / testsCount;

        std::cout << "Elements - " << testsCount << " Average case insertion - " << av-
erageTime << " ms" << std::endl;

        clocks = 0;
        time = 0;
        averageTime = 0;

        table.clear();

        long long int base = 1;
        while(base <= testsCount)
        {
            base <<= 1;
        }

        for(int i = 0; i < testsCount; i++)
        {
            long long int key = (i + 1) * base;
            int value = static_cast<int>(randomGen());

            clocks = clock();

            table.insert(key, value);

            clocks = clock() - clocks;
            time += static_cast<float>(clocks) / CLOCKS_PER_SEC;
        }
        averageTime = time * 1000 / testsCount;

        std::cout << "Elements - " << testsCount << " Bad case insertion - " << av-
erageTime << " ms" << std::endl << std::endl;
        averageTime = 0;

        testsCount *= 2;
    }

    return 0;
}
```

myhashtable.h

```
#ifndef HASHTABLE_H
#define HASHTABLE_H

#include<list>
```

```cpp
#include<iostream>
#include <sstream>

template<typename ValueT>
class HashTable
{
private:
    struct pair
    {
        long long int key_;
        ValueT value_;
    };

    int size_ = 8;
    int elementsCount_ = 0;
    std::list<pair>* array_ = nullptr;

    unsigned hash(const long long int& key) const;
    void resize(int size);

public:
    HashTable();
    ~HashTable();
    HashTable(const HashTable& table);
    void operator=(const HashTable& table);

    void insert(long long int key, ValueT value);
    ValueT get(const long long int& key) const;
    void remove(const long long int& key);
    bool search(const long long int& key) const;
    void clear();
    std::string toStr() const;
};

template<typename ValueT>
HashTable<ValueT>::HashTable()
{
    array_ = new std::list<pair>[size_];
}

template<typename ValueT>
HashTable<ValueT>::~HashTable()
{
    delete[] array_;
}

template<typename ValueT>
HashTable<ValueT>::HashTable(const HashTable& table)
{
    this->size_ = table.size_;
    for(int i = 0; i < size_; i++)
    {
        array_[i] = table.array_[i];
    }
}

template<typename ValueT>
void HashTable<ValueT>::operator=(const HashTable& table)
{
    this->size_ = table.size_;
    for(int i = 0; i < size_; i++)
    {
        array_[i] = table.array_[i];
    }
}

template<typename ValueT>
void HashTable<ValueT>::insert(long long int key, ValueT value)
{
    if(elementsCount_ >= size_)
```

```
    {
        resize(size_ * 2);
    }

    elementsCount_ += 1;

    unsigned hash_ = hash(key);

    std::list<pair>& valueList = array_[hash_];

    for(auto& elem : valueList)
    {
        if(elem.key_ == key)
        {
            elem.value_ = value;
            return;
        }
    }

    pair newPair{key, value};

    array_[hash_].push_back(newPair);
}

template<typename ValueT>
ValueT HashTable<ValueT>::get(const long long int& key) const
{
     unsigned hash_ = hash(key);

     std::list<pair>& valueList = array_[hash_];

     for(auto& elem : valueList)
     {
         if(elem.key_ == key)
         {
             return elem.value_;
         }
     }

     std::cout << "Can't find key: " << key << std::endl;
     throw "Key doesn't exist";
}

template<typename ValueT>
void HashTable<ValueT>::remove(const long long int& key)
{
    elementsCount_ -= 1;
    unsigned hash_ = hash(key);

    std::list<pair>& valueList = array_[hash_];

    for(auto iter = valueList.begin(); iter != valueList.end(); iter++)
    {
        if((*iter).key_ == key)
        {
            valueList.erase(iter);

            return;
        }
    }

    std::cout << "Can't find key: " << key << std::endl;
    throw "Key doesn't exist";
}

template<typename ValueT>
bool HashTable<ValueT>::search(const long long int& key) const
{
    unsigned hash_ = hash(key);
```

```
    std::list<pair>& valueList = array_[hash_];

    for(auto iter = valueList.begin(); iter != valueList.end(); iter++)
    {
        if((*iter).key_ == key)
        {
            return true;
        }
    }

    return false;
}

template<typename ValueT>
unsigned HashTable<ValueT>::hash(const long long int& key) const
{
    unsigned basicHash = abs(key) % size_;

    return basicHash;
}

template<typename ValueT>
void HashTable<ValueT>::resize(int size)
{
    int oldSize = size_;
    size_ = size;
    std::list<pair>* newArray = new std::list<pair>[size_];

    for(int i = 0; i < oldSize; i++)
    {
        for(auto& elem : array_[i])
        {
            unsigned hash_ = hash(elem.key_);

            pair newPair{elem.key_, elem.value_};
            newArray[hash_].push_back(newPair);
        }
    }
    delete[] array_;
    array_ = newArray;
}

template<typename ValueT>
std::string HashTable<ValueT>::toStr() const
{
    std::stringstream out("");
    for (int i = 0; i < size_; i++)
    {
        out << "Hash - " << i << std::endl;
        for(auto& elem : array_[i])
        {
            out << "----[" << elem.key_ << " - " << elem.value_ << "]" << std::endl;
        }
    }

    return out.str();
}

template<typename ValueT>
void HashTable<ValueT>::clear()
{
    size_ = 8;
    elementsCount_ = 0;

    delete[] array_;

    array_ = new std::list<pair>[size_];
}
#endif // HASHTABLE_H
```